

AffyExtensions

Ben Bolstad
bolstad@stat.berkeley.edu

October 25, 2002

Introduction

AffyExtensions is an R package created with the express purpose of expanding and improving some of the functionality of the affy package which is part of bioconductor. The goal of functions in this package is to be faster and less memory hungry than implementations in Affy. However, code in this package will be less extensible than code in the base package (affy). The package is loaded using

```
> library(AffyExtensions)
```

which will load the package and its requirements.

Computation of the RMA expression measure

The function supplied by this package (AffyExtensions) for computing the RMA measure is `RMA.C`. It is faster and less memory hungry than the similar functions `express` and `rma`. A typical interaction with the function would be

```
> my.data <- ReadAffy(cdffile, celfiles)
> my.exprs <- RMA.C(my.data)
```

which reads in the data and then computes RMA expression measures for all chips.

Speed comparisons with other implementations

To compare the speed of each of the three options we look at how long it takes to compute the expression measure by using output of `system.time`. The machine used for bench marking had the specifications given in the figure. Clearly other machines differ in specifications, but the general conclusions should hold.

The two items from `system.time` that we concentrate on are the user cpu and the elapsed time (all time in seconds). The results are shown in the figure.

Component	Specs
OS	Red Hat Linux 8.0
kernel	2.4.20-pre10-ac1 with preemptive patch applied
processor	AMD Athlon Thunderbird 1.2 Ghz
RAM	1 GB
Virtual	6 GB
R	R-1.6.0
affy	1.0.2

Figure 1: Machine Specifications

# Chips	RMA.C		express		rma	
	User CPU	Elapsed	User CPU	Elapsed	User CPU	Elapsed
4	13.38	14.87	175.05	180.40	161.54	165.77
5	16.32	18.15	176.01	181.31	170.17	175.11
10	29.57	32.75	267.91	275.30	235.55	242.21
15	43.97	47.48	306.66	315.46	295.91	304.33
20	57.27	61.41	400.90	412.51	361.03	371.78
25	71.49	75.50	469.12	494.82	415.94	429.14
30	84.58	88.87	556.40	668.55	506.63	533.74
40	112.36	118.45	722.68	757.33	661.61	721.29
50	139.83	148.19	872.11	955.85	776.45	863.26
60	167.95	177.63	1036.51	1547.79	927.12	1038.97
70	195.95	216.21	1184.65	1606.96	1058.32	1348.57
80	225.47	253.38	*	*	1197.41	1643.34
90	252.23	309.73	*	*	1343.75	2261.50
97	273.68	352.85	*	*	1430.34	2407.14

Figure 2: Results of timing simulation

Note the * signifies that I was unable to get the function to compute expression measures because we ran out of memory. The i32 architecture has a 3GB per process memory limit. To ensure that we are simulating real world conditions, each simulation was started in a fresh R session.

From the simulation we can make several conclusions

1. `RMA.C` is much faster than either of the other two methods
2. `express` is the most memory hungry and thus can deal only with the smallest of problem
3. `rma` is slow but less memory hungry than `express` (it still requires more memory than `RMA.C`)

How do the expression measure implementations differ?

The function `express` performs normalization before background correction. This is not the correct order to perform the operations. The function `rma` uses a background that uses only the PM probes. The function `RMA.C` uses a background that makes use of all probes on the chip. In general the differences between `rma`, `express` and `RMA.C` are small. Note that `RMA.C` orders its output slightly differently from `rma` and `express`.

The `RMA.C` also uses a fixed version of quantile normalization code, that deals better (and more correctly) with ties than the code in the `affy` package.

Quantile normalization code

The `AffyExtensions` package contains quantile normalization routines that mask those in the base package. They differ from the base package routines because they deal a little bit better with ties. In most cases this makes little difference to the computed expression measure, but it is a little better to fix it. The following code:

```
> normalize.quantiles(matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 1, 1, 1,
+ 2, 2, 6, 7, 8), 8, 2))
```

should (and with `AffyExtensions` does) yield

```
      [,1] [,2]
[1,]  1.0  1.5
[2,]  1.5  1.5
[3,]  2.0  1.5
[4,]  3.0  3.0
[5,]  3.5  3.0
[6,]  6.0  6.0
```

```
[7,] 7.0 7.0
[8,] 8.0 8.0
```

with the base affy package the (incorrect) result is

```
      [,1] [,2]
[1,] 1.0 1.0
[2,] 1.5 1.5
[3,] 2.0 2.0
[4,] 3.0 3.0
[5,] 3.5 3.5
[6,] 6.0 6.0
[7,] 7.0 7.0
[8,] 8.0 8.0
```